# Spark as Data Supplier for MPI Deep Learning Processes

Amir Haderbache[1,2,a)] Masahiro Miwa[2,b)] Masafumi Yamazaki[2,c)] Tsuguchika Tabaru[2,d)]
Kohta Nakashima[2,e)]

***Abstract:***
Recent works in deep learning show that training large models can improve accuracy. Many distributed deep learning frameworks have been so far developed to scale up machine learning algorithms. For the sake of performance, we believe these intensive computations must be combined with a clever data parallelism strategy. This paper brings one possible answer to the issue of supplying data to deep learning worker nodes on HPC systems. We design a two sides system where independent MPI Process executions match Spark tasks whose job is to provide data partition. We test and evaluate different Spark configurations and show that this system provides a flexible and scalable data supply mechanism which leverage MPI high performance and Spark high level data management.

## 1. INTRODUCTION

Nowadays big data are invading us in every aspect of our lives: signal from smartphones, stock market data or particle physics research are a few examples [1]. Machine learning applications which leverage these data tends to be a valuable business for companies as the exponential data growth is going to continue over the following years [2]. Recent learning algorithms, which require more computation power to handle very large data volume, need multiple computer nodes to be processed. We are convinced that developing distributed deep learning framework which combines high performance computation, clever data management and friendly user experience requires a collaborative use of HPC and Big Data Analysis tools.

However, because tools such as Hadoop and Spark were developed for a totally different purpose than traditional HPC tools like MPI, their mutual exploitation needs specific system designs. Indeed, MPI was firstly developed for distributed computing and is optimized to leverage multi-core architecture and shared memory processing. MPI is flexible, gives a total control on the application and allow both asynchronous as well as synchronous communication patterns. All of These attributes make MPI able to achieve high performance and really suitable for HPC requirements. Nevertheless, MPI is not really suitable for general user experiences targeted by deep learning applications.

On the other hand, Apache Spark and Hadoop are open-source data analytics frameworks designed to operate on datasets with a high level of abstraction such as Map-Reduce model [4]. MapReduce implementations such as Spark and Hadoop are designed to scale out data processing on commodity machines and provide scalability, fault-tolerance and high level programming interface. They are also very convenient for managing, supply or distribute huge datasets within a cluster of machines and perform operation on data in a easier way than MPI. However, the lack of control over communication details in MapReduce makes it less performant than low-level MPI implementation.

In Fujitsu Laboratories, we have enhanced the BVLC Caffe Framework [12] with MPI to distribute computation and decrease communication overhead by overlapping propagation steps with the aggregation of weight decays. MPI fine control provided good performance but the system needs to be enhanced with a data supply management. In order to provide resiliency, data staging, ability to express iterative algorithms with a high level and user friendly interface, we considered Apache Spark well-suited to reinforce such a software like the enhanced Caffe.

In this paper we describe our experiments to make Spark and MPI communicate to leverage MPI fine computing control and high performance with Spark high level data supply management for the purpose of deep learning application. The paper is organized as follow: in section 2 we briefly present Spark and MPI specificities, then Section 3 describes our system design enabling the inter-communication between MPI processes and Spark JVM. Section 4 explains the details of the system architecture. Then in Section 5, we present our experiments and evaluation results. Finally, we conclude in Section 6 about about the efficiency of such a system .

## 2. PRELIMINARIES

### 2.1 MPI

Message Passing Interface (MPI) is a language-independent communications specification for parallel computing where

[1] ESIEE PARIS, 2 Boulevard Blaise Pascal, 93162 Noisy-le-Grand, France
[2] FUJITSU LABORATORIES LTD., 4-1-1 Kamikodanaka, Nakahara-ku, Kawasaki, Kanagawa 211-8588, Japan
[a)] amir.haderbache@edu.esiee.fr
[b)] masahiro.miwa@jp.fujitsu.com
[c)] m.yamazaki@jp.fujitsu.com
[d)] tabaru@jp.fujitsu.com
[e)] nakashima.kouta@jp.fujitsu.com

point-to-point and collective communication are supported. MPI goals are high performance, scalability, and portability [5]. MPI is currently the dominant model used in high-performance computing and is a de facto communication standard that provides portability among parallel programs running on distributed memory systems [5]. MPI historically supported the initial growth of cluster computing and helped to shape what the computing world has become today with essential features like messages routines standard and collective operations. However, programming at the transport layer is obviously awkward for the handling of high level data structure such as distributed arrays, data frames, trees, or hash tables and tools like Spark are more suitable for this purpose.

### 2.2 Spark

Spark is a framework for general distributed computing which can handle Big Data using in-memory features with high level APIs for programming language like Scala, Python and Java. It translates the user program into a dataflow graph, mainly directed acyclic graph (DAG), and optimizes the program schedule for the data processing according to the dependency in this graph. Map-Reduce model is executed through the graph where *map* are local data processing and *reduce* global operation involving communication between nodes. Spark has its own job scheduler which leverages data locality and load balancing but it can also collaborate with other open source scheduler like Mesos [13] and Yarn [14]. Spark uses a data abstraction called *resilient distributed datasets* (RDD) [3] enabling data persistence and distribution. RDD can be cached in memory or stored in local storage. RDD also keep tracks of their original information even after being modified so that they may reuse their original values (RDD graph lineage)[3]. Since Spark adopts a *lazy evaluation* model, data are not processed until the user calls an *action* which returns the result of a computation. Before calling an *action*, the user defines a program with multiple *transformation*. The sequence of *transformation* creates a dataflow graph where new RDD are created from the older ones.

We should note that any Spark process working on a cluster is nothing else than a Java Virtual Machine (JVM) process. And as any JVM, a Spark process is allocated a Heap memory space with a capacity of 512 MB by default. Spark allow us the utilisation of 90% of the heap size (*spark.storage.safetyFraction* parameter) which is called the safe heap. Among this safe heap, some amount of memory is reserved for data caching and this part is usually 60% of the safe heap (*spark.storage.memoryFraction* parameter). So it is important to be aware that only (90% of 60/

## 3. MERGING SPARK AND MPI

### 3.1 Data Communication Path

Here we describe how the data communication between Spark JVM and MPI Process is performed in our system.

Since the Java Development Kit version 1.4, Java NIO API provides *MappedByteBuffer* which helps to establish a virtual memory mapping from JVM space to filesystem pages [8]. This removes the overhead of transferring and coping the files content from OS kernel space to JVM memory. Indeed, OS uses virtual memory to cache files outside of Kernel space to make them sharable with non-kernel processes. Java can maps the file pages to *MappedByteBuffer* directly and can process them without load into JVM. The idea is to make MPI Processes perform the same memory mapping by using the GNU C Library *mmap* function (Fig. 1). Through this mechanism, any MPI Process and Spark JVM can share memory, thus communicate data through virtual memory.

*MappedByteBuffer* maps directly with opened file in Virtual Memory by using the *map* method of a *FileChannel* object related to an opened file (Fig. 2). The *MappedByteBuffer* object works like a buffer but its data are stored in a file within the OS Virtual Memory. The *get* method on *MappedByteBuffer* object fetches data from file. In a similar way, the *put* method updates the content directly on the mapped file. Modified content is visible to other reader of the file. Processing file through *MappedByteBuffer* has big advantage because it doesnt make any *read-/write* system call on file, which improves the latency. Moreover, file in Virtual Memory caches the memory pages so that they may be directly accessed by *MappedByteBuffer* and thus does not consume JVM space.

```
int fd = open(filename, access, permission)
char* data = mmap(addr,length,prot,flags,fd,offset)
```

**Fig. 1** Memory Mapped File in C language used by MPI

```
RandomAccessFile file = new RandomAccessFile(
    filename,permission)
MappedByteBuffer out =file.getChannel().map(mode,
    position,length)
```

**Fig. 2** Memory Mapped File in Java used by Spark

Therefore, Spark can handle large data volume and prepare fresh batch sized partitions before sending them towards MPI Processes through this shared memory.

### 3.2 Notification Path

We describe here the system control mechanism performed in our system.

Most operating systems provide interprocess communication (IPC) allowing processes to share data and exchange messages [9]. Named Pipe has been used for the system notification path. Named Pipe provides a bidirectional data channel, a file on the file system, where many processes can read from and write to as a buffer and thus communicate with each other with simple short message. The general idea is to use the Named pipe to synchronize and control the communication while data exchanges are managed by the Virtual Memory mapped. This is definitely a generic system which can be used for many purposes, with many kind of processes and with different approaches to pass message through Named Pipe. In this paper, we describe an instance of such a system, as presented in Fig. 3), that we developed for the interaction between MPI and Spark. Spark acts as a server, supplying data to the client which requests them. MPI clients send

a data request to a Spark server through the Named Pipe. This triggers the server task : fetching the data, writting them within the Data Shared Memory, then notifying the client. When receiving the notification, MPI clients access the required data through shared memory. This execution can easily be improved by advanced pipeline using double buffer or ring buffer.

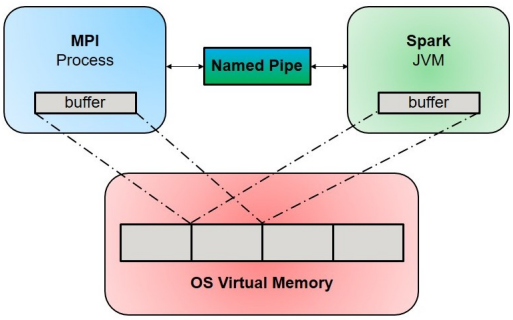


**Fig. 3** Spark-MPI with Memory Mapped File and Named Pipe

## 4. SYSTEM ARCHITECTURE

Our purpose is to build a scalable data storage system to feed MPI processes with data when required. Within the same cluster of machines, MPI processes should coexist with Spark workers. Both processes sets must be synchronized thank to the Named Pipe/Shared Memory we presented so far. The cluster nodes should be balanced in terms of process pairs. Each node hosts a Spark Executor where there are as many Spark cores allocated as there are MPI process running in the node. Then each Spark cores indirectly connects itself to one MPI process through a Named Pipe/Shared Memory system. It begins then a continuous client-server communication to supply data.

### 4.1 Data Supply Cycle

We implemented the communication system with MPI and Spark programs written respectively in C and Scala. Each of them take as parameters two configuration files listing the whole Mapped-file and Named pipes available that processes can use for their mutual communication. When the main program starts, each MPI process maps one Mapped-file and one Named-pipe listed from the configuration files.Then each opens its Pipe in a *WRONLY* (write only) mode and put a request in it. At this moment, MPI processes are blocked until Spark server read the request. On the server side, Spark main program does a similar thing: it assigns Named Pipe/Shared Memory location to each Spark core in the cluster. This assignments operation is explained in details in subsection 4.3. Then each Spark Task is able to open its pipe, in a *RDONLY* (read only) mode, and read the MPI requests written in it. Synchronously, after reading the request, each Spark reader unlocks one MPI process because it remains no other readers subscribed to the Named Pipe. Unblocked, the MPI process can reopen its Named Pipe, this time, in a *RDONLY* mode, and awaits the Spark notification answer (because nothing is written in the Pipe yet). Server side, after receiving the MPI request, Spark Task writes the requested data within the Mapped Shared Memory. We assume the data (RDD partitions) were already cached within the Spark Executor memory at this moment, following section describes in details the data storing and fetching from Database and Disk. The partition written within shared memory, Spark Taks reopens their respective Named Pipe in a *WRONLY* mode and insert a notification. The notification format is described later in the paper. At this moment, Spark server has finished its duty and can listen for another MPI request by reopening the Named Pipe in a *RDONLY* mode. Client side, the MPI processes are unblocked after receiving the notification. They can finally process the notification and read the specific partition from the shared memory. At this moment, the MPI client has been supplied and can ask for another partition by reopening the Named Pipe in a *WRONLY* mode. This cycle, presented in Fig. 4, is a traditional client-server communication between independent processes through a Named Pipe involving a mutual understanding on how to behave with the data stored inside the shared memory. Because each MPI client matches one distinct Spark Task, this system can be scalable with the number of nodes and cores.

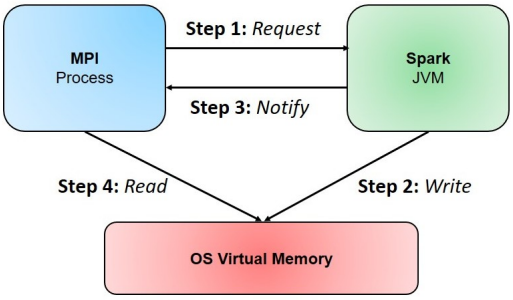


**Fig. 4** Cycle communication which supply data from Spark to MPI

### 4.2 Data fetching from LMDB Database

The details described in this section are explanation and how we used for our own purpose the *CaffeOnSpark* LMDB management [10]. When the Spark main program begins, it takes the database path given as a parameter where the data it has to supply to MPI processes are stored. Working on Deep Learning images processing with Caffe, we mainly handle *LMDB* database which provides a high-performance, embedded transactional database in the form of a *key-value* store. LMDB Databases may be used concurrently in a multi-threaded or multi-processing environment, with read performance scaling linearly. LMDB databases may have only one writer at a time, however unlike many similar key-value databases, write transactions do not block readers, nor do readers block writers. LMDB is also unusual in that multiple applications on the same system may simultaneously open and use the same LMDB store, as a means to scale up performance. Also, LMDB does not require a transaction log (thereby increasing write performance by not needing to write data twice) because it maintains data integrity inherently by design. LMDB was originally written in C but it has many API bindings for several programming languages. In our case we use *LMDB JNI*, a Java Interface for the LMDB library we can easily use with Spark to handle LMDB transaction [11]. By passing the *mdb* file path to a new *Env* object, we can call an *openDataBase* method which

returns a new *Database* instance. This instance enables so far to handle the Database with simple method called on it. The Spark program access to the Database thanks to this API. It begins to get the number of entries within the Database. An *entry* represents a single *key-value* pair recorded in the Database as Byte array. A program parameter indicates to the Spark program how we want to split the LMDB Database in many parts by outlining the number of *LMDB-Partition* chosen by the user. Knowing the number of entries in the Database and the number of LMDB-Partitions the user want, Spark can compute the number of entries it will assign per LMDB-Partition which defines the partition size. Spark creates so an Array of *LMDB-Partition* representing the whole Database. This Array is actually an Array of *spark.Partition* and its size is the number of partition desired by the user. Each Partition size is the total number of entries included in each LMDB-Partition. A Partition lists a set of keys where each key points to a specific value implemented as an Array of Bytes. The Fig. 5 describes this database management.

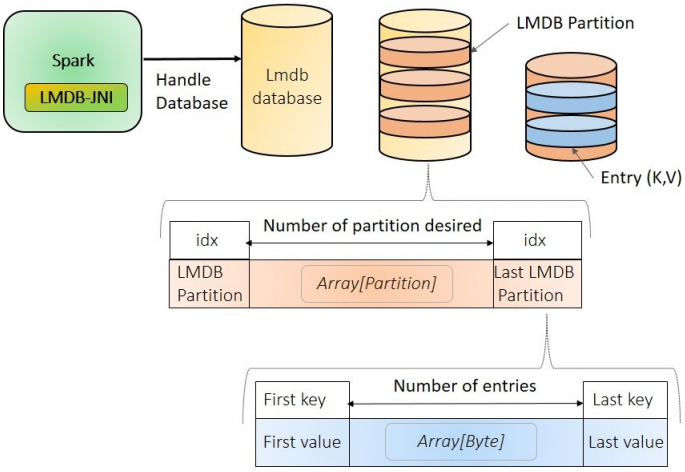


**Fig. 5** Spark partition an LMDB Database

To create a new kind of RDD, Spark developer needs to override the *getPartitions* and *compute* method of the RDD class. The *getPartitions* method should return an Array of Partition. As described a few lines before, this Array is constructed from an LMDB Database. Each Partition in this Array is identified with an index corresponding to the key of the first entry the Partition contains. The beginning of a Partition in the Database can be identify and knowing the partition size, the program can seek to another one. Once the Array of Partition constructed, the *compute* method defines a new iterator of the Database value. This is mandatory to build up an RDD where each partition is an LMDB-Partition and each element is an entry of the Database. Because our purpose is to perform a Deep Learning training, the LMDB Database entry values contains images data and related information for the Deep Learning Framework Caffe. Concretely, each value is an Array of Byte which contains : label, number of channels, height, width, encoded state of image and the image pixels themselves. The iterator made by the *compute* method goes through each LMDB-Partition index, then begin to process each entry inside the Partition. The Iterator get the entry key and the six data values listed previously, then it stores them inside a Scala Tuple of seven elements to wrap the entry into a single RDD element. This mechanism creates a new RDD whose the Partitions maps the LMDB-Partitions and where each element contains one images and its related information as a *Tuple* of seven element (A Tuple is a Scala data type containing a collection of various Scala Object). In our example, our Tuple looks like the following structure:(key:String, label:String, channel:Int, height:Int, width:Int, encoded:Boolean, data:Array[Byte]). The key is directly read with the *getKey* method of the *Entry* Object. Then it is stored inside a String. However, the value case is more complicated : the value is get from the *getValue* method, then it is passed into a *Protocol Language Message* builder which processes the value bytes and separate each information from the other. By this way, the iterator can get separately each information (label, witdth and so on) and store them in a specific scala variable for the *Tuple7*. The Fig. 6 shows in detail how the Spark RDD element are built from the database entries.

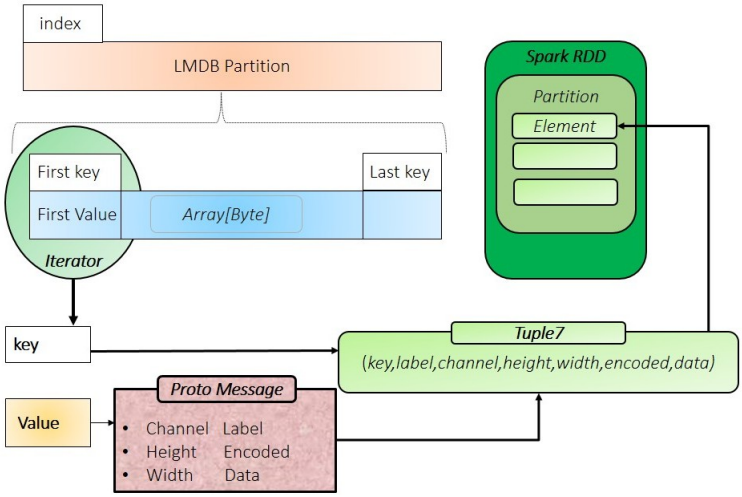


**Fig. 6** Spark compute the LMDB RDD

When a custom class can extends the Spark RDD class by overriding the getPartitions and compute methods, a serializable class can construct a sample RDD by overriding the *makeRDD* method. This method takes a *Spark context* as a parameter and should return an RDD. So by using a serializable class, overriding *makeRDD* by calling the custom RDD class constructor, the Spark main program can build up a new RDD from an LMDB Database. Next section explain how the Spark main program distributes the RDD partitions among the Executor cores in the cluster and assign to them a pair of path (Named Pipe and MappedFile) for communicating with MPI Processes.

### 4.3 Spark Tasks Control

Because RDD is a distributed collection of records, its datasets can easily be spread on multiple node and be processed in parallel. In our case, one partition of the RDD is attributed to one JVM cores and there is one JVM in a single node. RDD are distributed by design and the Partitions are the units of parallelism. First when we create the RDD, none execution occurs because RDD are lazy evaluated. Only action on the RDD can triggers the execution of a computation. Compare to MPI where the knowledge

of MPI process ranks and hosts enables to have a perfect control in how the process are spread, RDD spreading control is not really easy to manage. Spark tends to spread in a balanced way the partitions depending on the cluster resources.

In general, more numerous partitions allow the work to be distributed among more workers, but fewer partitions allow the work to be done in larger chunks, which may result in the work getting done more quickly as long as all workers are kept busy, due to a reduced communication overhead. Increasing partition number will make each partition to have less data. Spark can only run one concurrent task for every partition of an RDD, up to the number of cores in the cluster. So if we have a cluster with 50 cores, we might want our RDDs to at least have 50 partitions. The maximum size of a partition is ultimately limited by the available memory of an executor which is 54 % of the heap size allocated [6-7].

Performing an *action* into an RDD triggers the execution of RDD *transformation* and the parallel Task execution in the Executors. Spreading the RDD partitions across the cluster then make each Spark Executor core communicate with a specific MPI Processes is what we want. So we have to tell to each distinct executor core to connect itself with a specific Named Pipe and MappedFile. Usually, Spark perform the same action to each partition and it is not usual to assign specific task to each Executor core. However there are at least two ways to overcome this problem : the first one is working with *pairRDD* which are RDD's of Key-Value pairs. In Scala, a regular RDD can be easily turned into a pair RDD by running a *map* function returning key/values pairs, an example is given in Fig. 7. In our LMDB purpose, each *Tuple7* (a single element) can be splitted in two parts, the first one is the key and the second part is the six remaining values. This turns our RDD of *Tuple7* into a pairRDD where each element is a Scala *Tuple2* whose the first element is the key and the second element a Scala *Tuple6* containing the other values. Having a pairRDD, we can manipulate each element in it by referencing its key.

```
val pairsRDD = regularRDD.map(x => (x._1, (x._2,x._3,x._4,x._5,x._6,x._7)) )
```

**Fig. 7** Example of creating a *pairsRDD* with Spark Scala API

The second option is better in a sense it enables to manipulate directly each distinct partition and therefore each distinct Spark Executor core in the cluster. It use the *mapPartitionsWithIndex* function. When called upon an RDD, this function is called once for each Partition and not for each element in the RDD. This give us the advantage to program at the Partition point of view. When we call this function, we get an Iterator as an argument through which we can iterate through all element in this particular Partition. Last but not least, *mapPartitionWithIndex* provides also an index to track the Partition Number and therefore the Executor core identity. This function let the developer identify control a distinct core and therefore program a distinct task. Fig. 8 is code snippet showing how it is possible to use this function. The *index* represents the Partition ID, the *iterator* let us go through all the elements in a partition and *x* variable represents one element in the partition. This example show how we can associate to each element of a RDD, the corresponding ID of the partition it belongs to.

```
rdd.mapPartitionsWithIndex{ (index, iterator) => {
  val myList = iterator.toList
  myList.map(x => x + "␣->␣" + index).iterator}}
```

**Fig. 8** Example of using *mapPartitionsWithIndex* with Spark Scala API

Using this programing model, we can have a control upon each Spark allocated core and assign them a specific Named-Pipe/Shared Memory location by performing a mapping with the Partition index and configuration files lines.

### 4.4 Data Storage Management

This section describes how Spark manages the data storage while it supply MPI data request. Due to the large data volume the application handle, data are mainly stored into disk through the cluster. For the sake of performance, optimized Solid-state storage devices such as SSD *Nvme* Disk has been installed on each cluster node. As a Distributed File System, we use primary *HDFS*. We set up a YARN (Hadoop 2.0) cluster where the *DataNode* of each Hadoop slave stores their HDFS data blocks into *Nvme*. This is configured by turning the *dfs.data.dir* value of the *hdfs-site.xml* file into the mounted Nvme device location. Using then YARN as the Spark *master URL* when deploying the application (cf Fig. 9), Spark connects directly itself to the Hadoop cluster as presented in Fig. 10. This let any Spark Task to read and write data from/to HDFS. Spark on YARN maps each Spark Executor to a single YARN container and can consequently take advantage of HDFS data supports such as data locality, replication and fault tolerance.

```
spark-submit --class "Supplier" --master yarn --
    deploy-mode cluster --executor-memory $1 --num
    -executors $2 --executor-cores $3 file.jar
    arguments
```

**Fig. 9** Spark-submit command launching a Spark Application with YARN

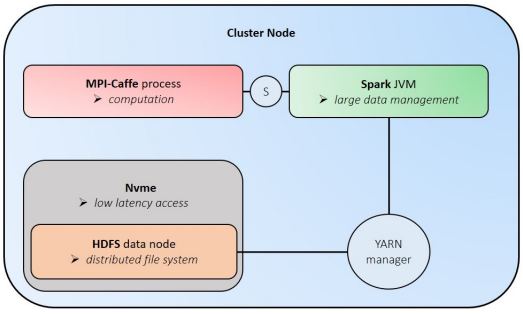


**Fig. 10** Spark connects itself to HDFS

The main Spark program handles the Database, splits it, creates an RDD, then assigns to each Executor through the cluster, one chunk of data (a certain number of partition according to the number of cores allocated to the Executor). One particular point

of our purpose is the fact that no one *Transformation* is performed over the RDD created from the Database. The reason is simple : we need to supply the entire Database entries to the MPI Processes. When the *mapPartitionWithIndex* function is launched, a first *action* is called upon the RDD. This triggers the beginning of the Database records processing by the Executor Tasks through the cluster. At this moment, two situations can happen depending on the size of the RDD partition compare to the cluster allocated resources. The first case is when the partitions are enough small to be entirely be cached in the Executor memory. This is the ideal case but it is unusual in practice because we process large data volume. So we have to establish a *persistence* strategy managing the case where RDD Partitions cannot be entirely held in Executor heap size memory. Hopefully Spark offers *RDD persistence* ability to store RDDs either in memory or hard disk or even a combination of both, with different levels of replication. RDD can therefore be marked to be persisted using the *persist* or *cache* methods on it as figured in Fig. 11. The first time the RDD is computed in an action, it will be kept in memory on nodes or spilled to Disk. Sparks *storage levels* are meant to provide different trade-offs between memory usage and CPU efficiency depending the following case :

- **RDD Partition fit entirely in JVM memory**
  Default storage level *MEMORY ONLY* is used. This is the most CPU-efficient option, allowing operations on the RDDs to run as fast as possible. However, we use this storage level only when we are pretty sure the entire set of partitions can be fit in memory. If not, some partitions will be recomputed on the fly each time they are needed, involving a computation time higher than if it will be read directly from optimized disk like Nvme (this is not always true and depend one the RDD computation and the specific disk used).
- **RDD Partition does not fit in memory**
  Storage level *MEMORY AND DISK* is used. It stores RDD as deserialized Java objects in the JVM like the previous case but, if the RDD Partitions do not fit in memory, the partitions that cannot be cached in memory are stored on Nvme. Then, when the Spark Task needs to process these partition, it will read them from Nvme. This mechanism is described in Fig. 12.

```
val rdd = database.makeRDD(sc)
rdd.persist(StorageLevel.MEMORY_AND_DISK)
```

**Fig. 11** Example of RDD creation and StorageLevel configuration with Spark Scala API

Spark automatically monitors cache usage on each node and drops out old data partitions in a least-recently-used (LRU) fashion. Once a partition has been written on the Data Shared Memory for being accessible to the MPI client, we don't use it anymore so the least-recently written partition will be automatically dropped out the moment when a Task will fetch a new Partition from Disk. An important note is **the maximum size of a partition is ultimately limited by the available memory of an executor.** Thus we have to be sure, before starting the application, that the size of a single partition is smaller or equal to the Executor memory allocated. Else, none partition might be cached, decreasing dramatically the performance.

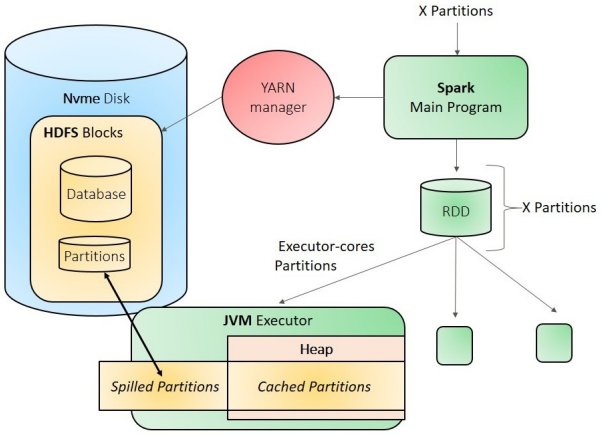


**Fig. 12** Data Storage Management

To manage allocated memory it is important to figure the Spark JVM memory model: *Spark memory* represents 75% of *Java Heap - Reserved Memory* [6-7]. The pool managed by Spark is split in two regions: *storage memory* and *execution memory*. The boundary between them is set by *spark.memory.storageFraction* parameter, which defaults to 0.5. The advantage of this memory management scheme is that this boundary is not static, and in case of memory pressure the boundary would be moved meaning that one region would grow by borrowing space from another one. *Storage Memory* pool is used for storing cached data and temporary serialized data. In case there is not enough memory to fit the whole unrolled partition, it would directly put it to the drive if persistence level allows it. *Execution Memory* pool is used for storing the objects required during the execution of Spark Tasks. This pool also supports spilling on disk if not enough memory is available.

Due to the nature of *Execution Memory*, data blocks cannot forcefully evicted from this pool, because this is the data used in intermediate computations and the process requiring this memory would simply fail if the block it refers to will not be found. But it is not so for the *Storage Memory* which is just a cache of blocks stored in RAM, and if we evict the block from there we can just update the block metadata reflecting the fact this block was evicted to Disk, and trying to access this block, Spark Task would read it from Disk (or recompute it in case persistence level does not allow to spill on HDD). Many case can occurs :

- We can forcefully evict the block from *Storage Memory*
- We cannot do it for *Execution Memory*
- If free space is available in *Storage Memory* pool (cached blocks dont use all the memory available), its size is reduced, increasing the *Execution Memory* pool one.
- If *Storage Memory* size exceeds its initial region (e.g, all the space is used by cached data), blocks are forcefully evicted untill the pool reaches its initial size.
- *Storage Memory* pool can borrow free space from *Execution Memory* if it is available.

Initial *Storage Memory* region size, is determined as

*Spark Memory* * *spark.memory.storageFraction* = (*Jav Heap - Reserved Memory*) * *spark.memory.fraction* * *spark.memory.storageFraction*. Default values give (*Jav Heap*-300MB) * 0.75 * 0.5 [6-7].

When we *cache* the partitions, we are pretty sure that the to tal amount of data cached on Executor is at least the same as th initial *Storage Memory* region size. However, if the *Executio Memory* region has grown up beyond its initial size before w filled the *Storage Memory* , we will not be able to forcefully evic entries from it, involving a smaller cache size. The tradeoff be tween caching data or spilling to Disk can be then managed b changing the parameter value depending on the application requirements.

### 4.5 Notification format

When the RDD is created from the Database and Partitions distributed, each Task needs to write the elements into shared memory, then send a notification to the client. However, because data written in shared memory are a kind of sequential bytes array, we need a way to distinguish internally, each partition and its different elements so that the MPI client can be able to read the right data at the right location. To fix this issue, we designed a set of rules making MPI clients understand the way Spark server write the data . First, the Spark Task access to each element of its Partition (e.g Scala Tuple7) and converts each different value into Bytes array (creating 7 distinct bytes array). The size of each value, as a bytes array, is then recorded and used later. Next the seven arrays are concatenated together to build up a single array byte representing the entire element. Spark server is able to read distinctly each value from this contiguous array because it know their size and the general order they have been concatenated inside the array. The size of the global array is also recorded and the array is written in shared memory. Doing this preliminary job is necessary to send a proper notification through the Named Pipe and make the client able to access the right data. The notification is composed of the total size of the array written in memory, the size of each different values encoded within the array and the offset position in the shared memory where the array data actually begins. On receiving this notification, the MPI Process is able to read the particular area from the shared memory where the next Partition element is written . Because it knows the size of each element value, it can extract any particular images metadata and the image itself. By notify only the data size and position, independent Spark and MPI processes can understand each other. The sequence of these operations are visible in Fig. 13.

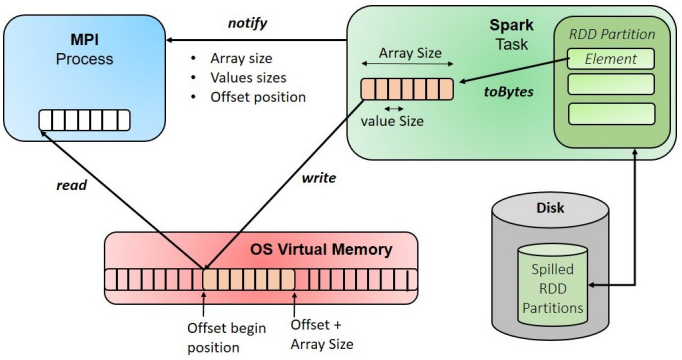


**Fig. 13** Notification management

## 5. SYSTEM PERFORMANCE

### 5.1 Experimental Setup

We managed our experiments on two compute nodes (*server#1*, *server#2*). Each node has a Dual sockets of 2.6 GHz 14-core Intel "Xeon" E5-2697 v3 processors. Both node are connected by InfiniBand EDR (Mellanox ConnectX-4). They have each a 128GB DDR4 SDRAM memory module and are also equipped by a 1.2TB Intel SSD "Nvme" which has a theoretical sequential read performance up to 2.6 GB/s and 1.6 GB/s for write operation. The experiments have been conducted under Linux Centos7 OS. We used OpenMPI 1.7.4 as the MPI implemetation and we ported Spark 1.6.1 to run on the two nodes. Spark was deployed with a YARN 2.6.1 master to facilitate the control of the resources allocated in each node. One *HDFS DataNode* per node were running and only one *NodeManager* on *server#1*. Spark Application was launched in client-mode from *server#1*, (e.g the Spark driver runs directly on *server#1*) with 5G allocated to the *Spark driver*. To improve the Spark data management performance, we also used a fast serializer such as *KryoSerializer*. The heap memory allocated to each Spark Executor depended on the Database size and the choice of the number of partitions. At least this memory should had been higher than the partition size. We used 2 Executors, one for each node (the number of Executor maps the number of compute node), and 8 local cores were allocated to each Executor (see below result to understand why). This means that each Executor can run 8 parallel Tasks at the same time where each Task process one distinct partition. So in order to make each core always running and avoid CPU idleness, the memory allocated to each Executor had to be at least eight times the partition size. By this way, 8 partitions can be cached in-memory and none core stay idle. When a partition has been processed (e.g written into the shared memory), a core can catch the next one from Disk as explained in last section.

### 5.2 MPI Processes Read from Data Shared Memory

To calibrate the global supply system performance, we evaluated distinct cycle step on a single node and figured out the configuration reaching to the best performance result.

First we made a Spark application write data and send notification to MPI Process and we measured its performance reading continuously a 1GB partition from Data shared memory using

*memcpy* function. The Fig. 14 shows that a single MPI client can achieve a bandwidth of 6.57 GB/s for reading data from the shared memory. Using 5 MPI clients achieves 33,55 GB/s. Because each MPI process is linked with a distinct Shared Memory, we can see that the performance results are scalable and almost proportional to the number of Process used.

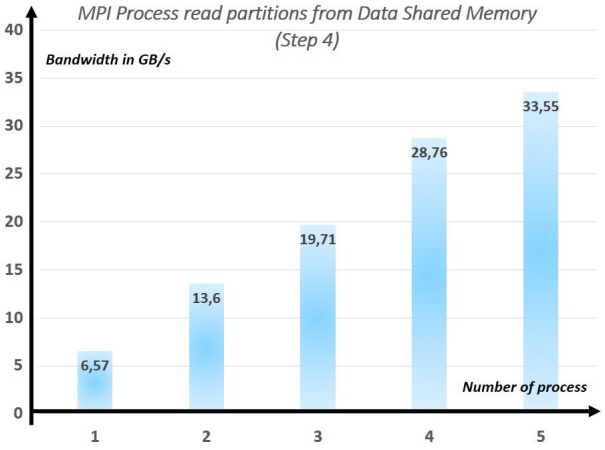


**Fig. 14** MPI Processes read from memory performance

### 5.3 Spark Write into Data Shared Memory

Once the RDD created and the Partitions spread. We measured a Spark core writing performance. Having its 1GB partitions cached entirely in memory, the Spark server continually writes them into shared memory using the *put* method of *MappedByteBuffer* object. Fig. 15 shows that a single Executor core achieves a bandwidth of 4.8 GB/s which is decupled by allocating more cores and more heap memory to the Executor, with 5 cores we reach 21.55 GB/s.

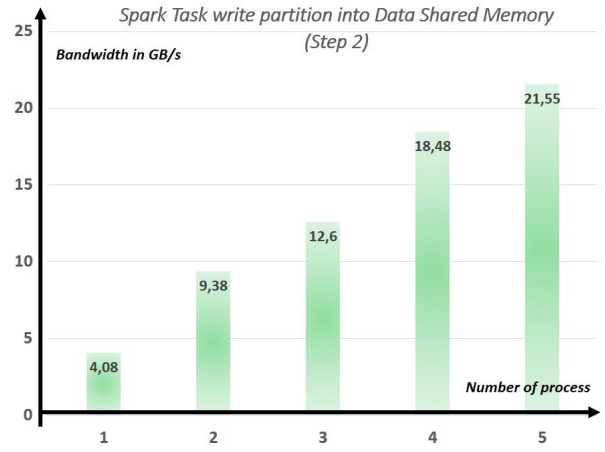


**Fig. 15** Single Spark Task write into memory performance

### 5.4 In memory Cycle Performance

We present here the evaluation of the system full cycle. By ”cycle”, we mean the whole step during the supply of data, from the MPI client request to the moment when it has read the desired partition from the shared memory after having received the server answer. The following evaluation has been made with Spark Executor caching the entire set of Partition after the RDD creation. This means that Spark Task did not need to read data from disk since the partition requested has been already prefetched into the heap. These results presents thus the In-Memory cycle performance without spilled Partitions. Fig. 16 shows the bandwidth of the supply system with a single node, related to different number of pairs of client/server. The bandwidth emphasizes the rate at which the Spark JVM supply data to MPI client. One pair means one independent MPI client communicating with one distinct Spark Task.

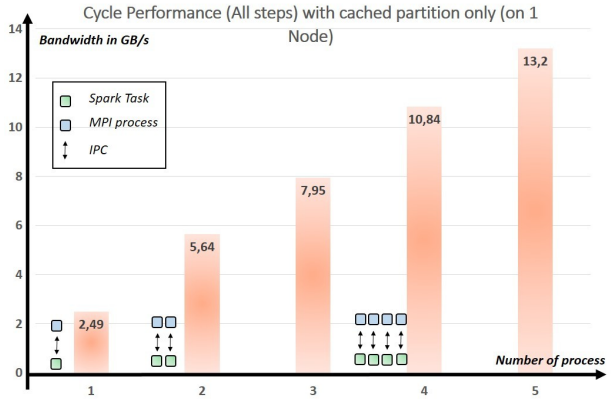


**Fig. 16** 1 Node, Cycle Performance

Fig. 17 describes the same experiment than the previous one using two nodes rather than one. We can see that the results with 2 node are quite twice of the results given with one. This shows that the system can be scale out according to the number of node used.

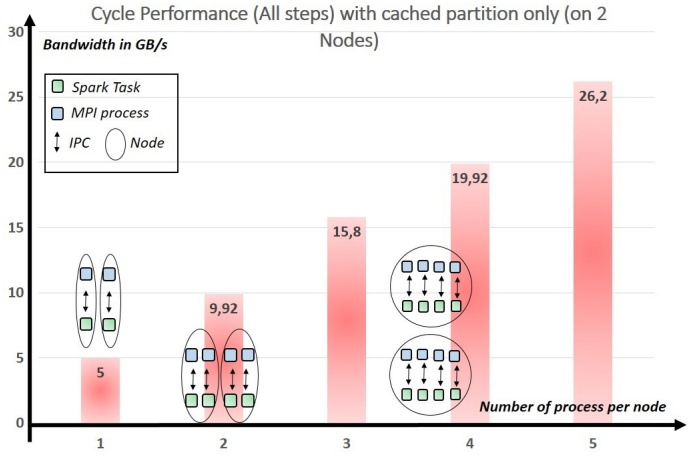


**Fig. 17** 2 Nodes, Cycle Performance

### 5.5 Spilled Data Latency from Nvme

We measured then the performance of Spark server reading data stored in Nvme. This situation happen when Spark Task need to access to a new Partition which is not cached in memory but has been spilled on Disk. We launched the same Spark Application reading data from Disk with different number of cores.

Fig. 18 reveals that using at least 8 Spark cores is sufficient to reach the theoretical maximum value of Nvme bandwidth which is 2.5 GB/s. This results provides us so far with the optimal number of core (e.g. 8) we had to allocate per Executor in order to reach the best performance for our cluster.

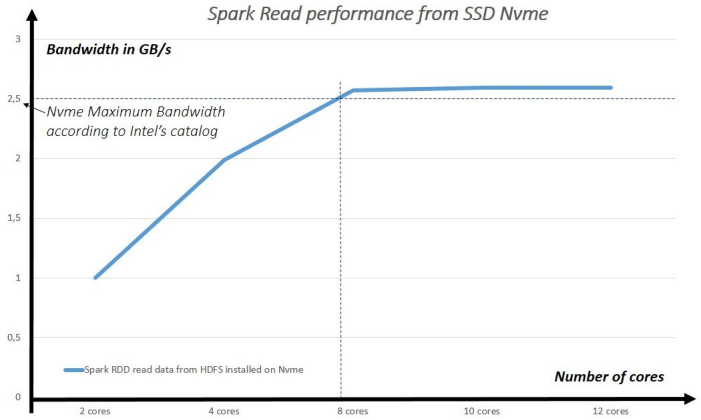


**Fig. 18** Spark Tasks read spilled Partitions in Nvme

### 5.6 Cycle Performance with Spilled Data

Fig. 19 describes the partition processing during a cycle data supply along the execution of the application. In the begining, the system supplies the partition already cached in memory (the 25 first partition are cached whereas the other are evicted on Nvme), then it begins from the 26th partition to read partition from Disk. This involves a significant difference in the speed performance because the data latency increases. Also, the more we increase the number of processes, the more the bandwidth is high. The performance of such a system is clearly limited by the Disk I/O but we can reach better performance with multiple independent processes.

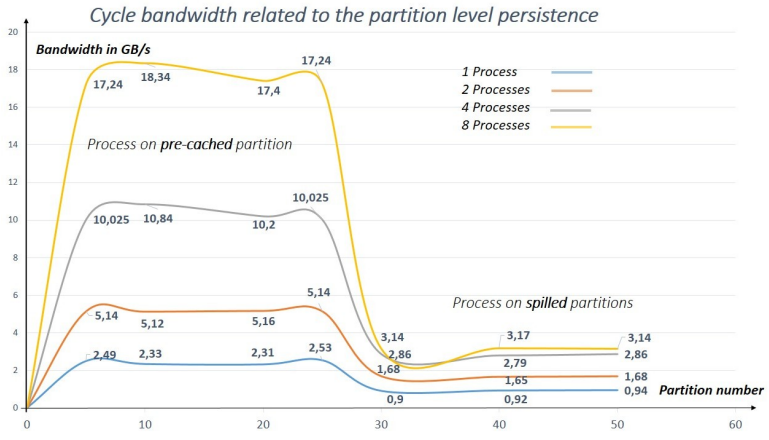


**Fig. 19** Cycle Bandwidth

## 6. CONCLUSION

We designed and evaluated an efficient way to make Spark JVM and MPI Process exchange data while staying independent from each other. Sharing data using Memory Mapped File gives the advantage of not relying on read-write system calls and Named pipe brings an easy way to synchronize processes during the execution. Letting Spark to be in charge of the data management (fetching, preparation, distribution and persistence) across the cluster facilitates greatly the Big Data processing due to the high level programming interface it provides. The traditional HPC tools such as MPI can then be used entirely to perform high performance computation and request already prepared data only when necessary. *HPC becomes a client for Big Data server.* The purpose of this system is obviously to bring a general way to share data. Therefore it can be use for many kind of applications. The LMDB Database handling and images partitions processing give an example on how we can use this system to supply Deep Learning worker node to scale out Neural Network training. The evaluation results show how the system can scale and how the supply bandwidth can be improved with the resources (cores and memory) allocated to each Executor enabling us to reach a value of 26.2 GB of data supplied with 10 processes. Our experiments also emphasize how this system is only limited by the disk I/O bottleneck. Future work will be focused on how we can overcome the communication overhead in Spark control to leverage the maximal performance of disk I/O.

### References

[1] Y. Yiteng Zhai Ong, and I Tsang *The emerging big dimensionality*, IEEE Computational Intelligence Magazine, (2014)

[2] S. Mills, S. Lucas, L. Irakliotis, M. Rappa, T. Carlson, and B. Perlowitz. *Demystifying big data: a practical guide to transforming the business of government*. In Technical report, available from ⟨http://www. ibm. com/software/data/demystifying-big-data⟩, pages 1-100, 2012.

[3] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, J. Michael, Franklin Scott Shenker, and Ion Stoica *Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing*. In USENIX Conference on Networked Systems Design and Implementation, pages 2-2 (2012)

[4] Tom, White *Hadoop: The definitive guide*. OReilly Media Inc. (2012)

[5] MPI Forum community *MPI: A Message-Passing Interface Standard* Version 3.1, Message Passing Interface Forum, June 4, 2015. available from ⟨http://www.mpi-forum.org. Retrieved on 2015-06-16.⟩

[6] Alexey Grishchenko *Distributed Systems Architecture* available from ⟨https://0x0fff.com/spark-memory-management/⟩

[7] *Apache Spark Official Website : Memory Management Oveview* available from ⟨http://spark.apache.org/docs/latest/tuning.html/memory-management-overview⟩

[8] Nitin *JAVA NIO Memory-Mapped File* Posted on November 5, 2014 available from ⟨https://tutorials.techmytalk.com/2014/11/05/java-nio-memory-mapped-files/⟩

[9] Stevens, Richard. *UNIX Network Programming, Volume 2, Second Edition: Interprocess Communications.* Prentice Hall, 1999. ISBN 0-13-081081-9

[10] Yahoo! Anfeg *CaffeOnSpark, Spark for Deep Learning* available from ⟨https://github.com/yahoo/CaffeOnSpark⟩

[11] deephacks *LMDB JNI, a Java Interface for the LMDB library* available from ⟨https://github.com/deephacks/lmdbjni⟩

[12] BVLC Caffe *Deep learning framework by the BVLC* available from ⟨http://caffe.berkeleyvision.org/⟩

[13] Apache Mesos *A distributed systems kernel* available from ⟨http://mesos.apache.org/⟩

[14] Apache YARN *Apache Hadoop NextGen MapReduce* available from ⟨https://hadoop.apache.org/docs/r2.7.1/hadoop-yarn/hadoop-yarn-site/YARN.html⟩